Symbolic Techniques for Formally Verifying Industrial Systems

Sérgio Campos Edmund M. Clarke Marius Minea

June 18, 1996

CMU-CS-96-148

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in: *Science of Computer Programming*, Elsevier Science

Abstract

The design of correct computer systems is extremely difficult. However, it is also a very important task. Such systems are frequently used in applications where failures can have catastrophic consequences, or cause significant financial losses. Simulation and testing are the most widely used verification techniques, but they can only show the presence of errors and cannot demonstrate correctness. Until lately formal methods were too expensive to be used in industrial problems, but recent research has made it possible to apply formal techniques to the verification of complex real-world systems. *Symbolic model checking* is an example of such a technique that has been successful in verifying large finite-state systems. It has also been extended to produce timing and performance information. These properties are extremely important in the design of high-performance systems and time-critical applications. A more detailed analysis of a model is possible using these extensions than by simply determining whether a property is satisfied or not. We present algorithms that determine the exact bounds on the delay between two specified events and the number of occurrences of another event in all such intervals. To demonstrate how our method works, we present two complex examples: the verification of the Futurebus+ cache coherence protocol and the timing analysis of the PCI local bus. These results show the usefulness of symbolic model checking in analyzing modern industrial designs.

This research was sponsored in part by the National Science Foundation under grant no. CCR-9217549, by the Semiconductor Research Corporation under contract 96-DJ-294, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330.

Keywords: real-time systems, formal verification, symbolic model checking, quantitative timing analysis, Futurebus^{+}, PCI Local Bus

1 Introduction

The task of verifying if a computer system satisfies its specifications is extremely important. For systems used in safety-critical applications, a failure to satisfy a property can have serious or sometimes even fatal consequences. Even in the case of non-critical systems, errors can cause significant financial losses and can be very difficult to correct.

Moreover, the inherent complexity of modern designs makes their analysis and verification a very difficult task. The most widely used methods for analyzing systems are simulation and testing. However, both methods can only show the presence of errors, and cannot establish correctness. Formal methods, on the other hand, can demonstrate that a system is correct, but they are more expensive to use, both in terms of time and effort spent for verification. Until recently such methods could not be applied to industrial problems, due to the complexity of these systems.

Advances in the research of formal methods as well as in the computing resources available for verification have changed this situation. Techniques such as symbolic model checking [3, 11] can now be used to solve problems of industrial complexity. Symbolic model checking is a technique for verifying finite-state hardware systems that can handle extremely large state spaces efficiently. It determines automatically if a system satisfies its specifications. Models with up to 10^{30} states can often be verified in minutes. The method has been used successfully to verify a number of real-world applications.

To demonstrate how symbolic model checking can be applied we describe the verification of the Futurebus+ cache coherence protocol, adopted as a standard by both IEEE and the U.S. Navy [6]. A precise model of the protocol has been constructed and a formal specification of cache coherence has been verified. This analysis uncovered a number of errors and ambiguities in the protocol that were not previously known. We believe that this is the first time that formal methods have been used to find nontrivial errors in a proposed IEEE standard.

This work also presents extensions to symbolic model checking that allow performance evaluation and quantitative analysis of time-critical applications. Timing information is extremely important when designing high-performance systems, or when trying to improve or maximize resource utilization. In addition, verifying performance guarantees becomes necessary when real-time applications are being analyzed. In this case, it is imperative that the performance claims be substantiated with a formal analysis that covers all possible executions.

Traditional formal verification algorithms assume that timing constraints are given explicitly in some notation like temporal logic. Typically, the designer provides a constraint on response time for some operation, and the verifier automatically determines if it is satisfied or not. These techniques do not provide any information about how much a system deviates from its expected performance, although this information can be extremely useful in fine-tuning the behavior of the system.

In this paper we give algorithms to compute quantitative timing information, such as exact upper and lower bounds on the time between a request and the corresponding response. We also describe algorithms that compute the minimum and maximum number of times a certain condition is satisfied on all paths between two given events. For example, we can use these algorithms to bound the time between asserting a bus request and receiving the corresponding bus grant. In addition, we may need to compute the number of times a third event occurs within such an interval, such as the number of times other transactions are issued between the bus request and the corresponding grant.

Our algorithms provide insight into *how well* a system works, rather than just determining whether it works at all. They enable a designer to determine the timing characteristics of a complex system given the timing parameters of its components. This information is especially useful in the early phases of system design, when not all parameters have been fixed. In this case, the information provided by our algorithms can be used to establish how changes in a parameter affect the global behavior of the system.

We use these techniques to analyze the performance of the PCI Local Bus. PCI is a high performance bus architecture designed to become an industry standard for current and future high-performance systems. It is used primarily in the Intel Pentium based systems, as well as in the DEC Alpha processor systems. We model the PCI bus, concentrating on its temporal characteristics, and analyze its performance. We compute transaction response time in various configurations of the system. We calculate bounds on the response time of a PCI transaction and produce detailed information about each phase of the communications protocol. This type of information allows the designers to understand the behavior of the system more accurately than the information generated by traditional verification methods.

The remainder of the paper is organized as follows. Section 2 presents the underlying theory for symbolic model checking with binary decision diagrams. In Section 3 the verification of the Futurebus+ cache coherence protocol is briefly described. The symbolic algorithms for computing the minimum and maximum length of the paths between two state sets as well as the algorithms for counting the number of states that satisfy a given condition along a path between two sets of states are described in section 4. Section 5 discusses the modeling of the PCI bus and section 6 shows how it can be analyzed using our techniques. Section 7 concludes the paper.

2 Symbolic Model Checking

Temporal logic model checking is a method for determining the correctness of finite-state systems. In this technique, specifications are written as formulas in a propositional temporal logic and computer systems are represented by state-transition graphs. Verification is accomplished by an efficient breadth first search procedure that views the state-transition graph as a model for the logic, and determines if the specifications are satisfied by that model.

There are several advantages to this approach. The state space search is performed completely automatically. Moreover, the model checker provides a counterexample if the formula is not true. The counterexample is an execution trace that shows why the formula is false. This is an extremely useful feature because it can help locate the source of the error and speed up the debugging process.

Another advantage is the ability to verify partially specified systems. If a component hasn't been fully specified, some of its outputs can be assigned nondeterministic values. The set of behaviors modeled this way is a superset of the actual behaviors of the component. Useful information about the correctness of the system can be gathered before all the details have been determined. The abstracted model is then refined when more information about the component becomes available. This allows the verification of a system to proceed concurrently with its design. Consequently, verification can provide valuable hints that will help designers eliminate errors earlier and define better systems.

An important characteristic of the model checking approach is that by using a finite-state model it is amenable to efficient implementations using symbolic techniques. In this approach the transition relation is represented by boolean formulas, and implemented by *binary decision diagrams* [1]. This usually results in a much smaller representation for the transition relation, allowing the verification of models several orders of magnitude larger than those verified using traditional implementations.

2.1 Binary Decision Diagrams

Binary decision diagrams (BDDs) are a canonical representation for Boolean formulas [1]. A BDD is obtained from a binary decision tree by merging identical subtrees and eliminating nodes with identical left and right siblings. The resulting structure is a directed acyclic graph rather than a tree. This allows nodes and substructures to be shared. The vertices of the graph are labeled with the variables of the Boolean formula, except for the two "leaves" which are labeled with 0 and 1. To insure canonicity, a strict total order is placed on the variables as one traverses a path from the "root" to a "leaf." The edges are labeled with 0 or 1. For every truth assignment there is a corresponding path in the BDD such that at vertex x, the edge labeled 1 is taken if the assignment sets x to 1; otherwise, the edge labeled 0 is taken. If the path ends in the "leaf" labeled 0, then the assignment does not satisfy the formula, and conversely, if the "leaf" reached is labeled 1, then the formula is satisfied by the assignment. Figure 1 illustrates the BDD for the Boolean formula $(a \wedge b) \vee (c \wedge d)$.

In [1], Bryant shows that given a variable ordering, the BDD for a formula is unique. The paper also gives efficient algorithms for computing the BDDs for $\neg f$ and $f \vee g$ given the BDDs for f and g. For the purposes of symbolic model checking, it is also necessary to quantify over Boolean formulas. Bryant describes an algorithm for computing the BDD of a restricted formula such as $f\mid_{v=0}$ or $f\mid_{v=1}$. The BDD for the formula $\exists v[f]$, where v is a Boolean variable and f is a Boolean formula, can be computed by $f\mid_{v=0} \vee f\mid_{v=1}$.

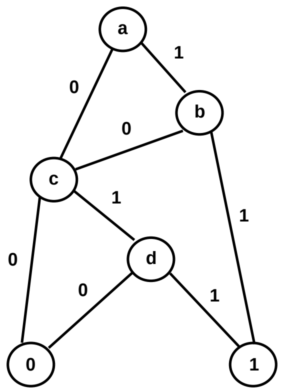

Figure 1: BDD for $(a \wedge b) \vee (c \wedge d)$

All of the formulas used in our algorithms are represented by BDDs. The BDDs for these formulas are built up in a bottom-up manner. The set of atomic propositions in these formulas is precisely the set of state variables, therefore the BDD for an atomic proposition consists simply of a single BDD variable. Since a formula is built up from atomic propositions using Boolean connectives, the BDDs for a formula can be constructed using the BDD operations discussed in the previous paragraph. In fact, the implementation allows arbitrary state formulas of computation tree logic (CTL) [5]. These formulas may contain branching time operators as well as logical connectives, but for the sake of simplicity, this discussion is limited to Boolean formulas.

2.2 Symbolic Representation of Transition Graphs

The system being verified is represented as a state-transition graph. States are labeled by atomic propositions, and a boolean variable is created to represent each proposition. An assignment of values to these variables defines a state in the graph (we assume that different states have different labels as described in [11]). For example, if the model has three propositions a, b, and c, examples of states are (a, b, c), $(\overline{a}, \overline{b}, c)$, and $(a, \overline{b}, \overline{c})$, where, for variable v, v means the variable is true in the state, and $\overline{v}$ means the variable is false. Boolean formulas over variables of the model can be true or false in a given state. The value of a boolean formula in a state is obtained by substituting into the formula the values of the variables in that state. For example, the formula $a \vee c$ is true in all states shown above. The graph representation used by our algorithms is a direct consequence of this observation. We use a boolean formula to denote the set of states in which that formula is satisfied. For example, the formula *true* represents the set of all states, the formula *false* represents the empty set of states, and the formula $a \vee c$ represents the set of states in which a or c are true. Because symbols are used to represent states, algorithms that use this method are called symbolic algorithms.

Transitions can also be represented by boolean formulas. A transition $s \rightarrow t$ is represented by using two sets of variables, one set for the current state and another set for the next state. Each variable in the next state set corresponds to one variable in the current state set. If s is represented by the formula f_s over the current state variables, and t is represented by the formula f_t over the next state variables, then the transition $s \to t$ is represented by $f_s \wedge f_t$. For example, a transition from state $(\overline{a}, \overline{b}, \overline{c})$ to state $(\overline{a}, b, \overline{c})$ is represented by the formula $\neg a \wedge \neg b \wedge \neg c \wedge \neg a' \wedge b' \wedge \neg c'$. The transition relation of a graph is constructed from the disjunction of all transitions in the graph. The meaning of the formula representing the transition relation is the following: there exists a transition from state s to state t iff the substitution of the variable values for s in the current state variables and of those of t in the next state variables of the transition relation yields *true*.

In the same way as boolean formulas can represent sets of states, they can also represent sets of transitions. Symbolic model checking takes advantage of this fact by grouping sets of transitions into a single formula, which often significantly simplifies traversing the graph. The clustering of transitions happens automatically when boolean formulas are implemented using BDDs. This occurs because of the canonicity of BDDs: given a fixed variable ordering, a boolean formula is represented by a unique BDD. Therefore, the order in which the transition relation is constructed does not affect the final result, the canonicity property guarantees that the same transitions will be clustered according to the formulas that represent them. This technique is one of the main reasons for the efficiency of symbolic algorithms.

2.3 Computation Tree Logic

The properties to be verified by the model checker are expressed in computation tree logic, CTL [5]. Computation trees are derived from state transition graphs. The graph structure is conceptually unwound into an infinite tree rooted at the initial state. Paths in this tree represent all possible computations of the program being modelled. Formulas in CTL refer to the computation tree derived from the model. CTL is classified as a *branching time* logic, because it has operators that describe the branching structure of this tree.

Formulas in CTL are built from atomic propositions (in our method, each proposition corresponds to a state variable in the model), boolean connectives $\neg$ and $\wedge$, and *temporal operators*. Each operator consists of two parts: a path quantifier followed by a temporal operator. Path quantifiers indicate that the property should be true of *all* paths from a given state (**A**), or *some* path from a given state (**E**). The temporal operators describe how events are ordered with respect to time for a path specified by the path quantifier. They have the following informal meanings:

- **F** φ (φ holds sometime in the future) is true of a path if there exists a state in the path that satisfies φ.
- **G** φ (φ holds globally) is true for a path if φ is satisfied by all states on the path.
- **X** φ (φ holds in the next state) means that φ is true in the next state of the path.
- φ **U** ψ (φ holds until ψ holds) is satisfied by a path if ψ is true in some state in the path, and in all preceding states, φ holds.

Bounded versions of the temporal operators exist [7]. They allow the expression of time-bounded properties, which can be used to verify the real-time behavior of systems [4].

Some examples of CTL formulas are given below to illustrate the expressiveness of the logic.

- $\mathbf{AG}(req \rightarrow \mathbf{AF}\ ack)$: A request is always followed by an acknowledge.
- $\mathbf{AG}(req \rightarrow \mathbf{AF}_{\leq 5}\ ack)$: A request is always followed by an acknowledge within less than 5 steps.
- $\mathbf{EF}(started \wedge \neg ready)$: It is possible to get to a state where $started$ holds but $ready$ does not hold.
- $\mathbf{AG}\ \mathbf{EF}\ restart$: From any state it is possible to get to the $restart$ state.
- $\mathbf{AG}(send \rightarrow \mathbf{A}[send\ \mathbf{U}\ recv])$: It is always the case that if $send$ occurs, then eventually $recv$ is true, and until that time, $send$ must remain true.

The model checking algorithm for CTL formulas determines the states in which a formula φ is satisfied. It computes in a bottom up fashion the set of states in which each sub-formula of φ is satisfied. Handling atomic propositions and logical connectives is straightforward. To check the formula **EX** φ we use an *image computation* [2] to find the states that have a successor satisfying φ. For the formula **EF** φ, we use a fixed point characterization of the temporal operator:

$$\mathbf{EF}\ \varphi = \varphi \vee \mathbf{EX}\ \mathbf{EF}\ \varphi$$

The fixed point is computed iteratively, starting from the empty set of states. Other temporal operators are handled in a similar way.

2.4 The SMV specification language

The system being verified is described in the SMV language. The language allows the specification of synchronous or asynchronous, detailed deterministic or abstract nondeterministic finite state machines. The language provides modular hierarchical descriptions, reuse of components, and parameterization so that multiple instances of a module can use different data values. Within every module, local variables may be declared. The type of a variable may be boolean, an enumeration type or an integer subrange.

```
VAR state0: {noncritical, trying, critical};
```

The value of the variables in each state are defined using `init` and `next`:

```
init(state0) := noncritical;
next(state0) :=
case
  (state0 = noncritical) : {trying,noncritical};
  (state0 = trying) & (state1 = noncritical): critical;
```

```
  (state0 = trying) & (state1 = trying) & (turn = turn0):
                             critical;
  (state0 = critical) :    {critical,noncritical};
  1: state0;
esac;
```

An SMV program can be viewed as a system of simultaneous equations whose solution determines the next state. When describing communication protocols, asychronous circuits, or other systems whose actions are not sychronized, we can define a set of parallel processes whose actions are interleaved arbitrarily in the execution of the program. Although primarily designed to describe circuits, the SMV language can be used to describe other types of systems like real-time systems and communication protocols. More information about the language can be found in [11].

3 The Futurebus Cache Coherence Protocol

This section briefly presents the formalization and verification of the cache coherence protocol described in the draft Futurebus+ standard (IEEE Standard 896.1–1991) [8]. The protocol has been modeled using the SMV language and a formal specification of cache coherence has been verified using symbolic model checking. Several errors and ambiguities have been found in the process. This experience demonstrates that hardware description languages and model checking techniques can be used to help design real industrial standards. A complete description of this model and its analysis is outside the scope of this paper, and can be found in [6].

Futurebus+ is a bus architecture for high-performance computers. The goal of the committee that developed Futurebus+ was to create a public standard for bus protocols that was unconstrained by the characteristics of any particular processor or device technology and that would be widely accepted and implemented by vendors. The cache coherence protocol used in Futurebus+ is required to insure consistency of data in hierarchical systems composed of many processors and caches interconnected by multiple bus segments. The Futurebus+ protocol maintains coherence by having the individual caches *snoop*, or observe, all bus transactions. Since all processors observe all transactions, accurate information about the status of a cache line is updated at each processor every time requests for that cache line are issued. This information is used to preserve data integrity in the presence of multiple requests. Coherence across buses is maintained using *bus bridges*. They act as proxies of processors in other levels of the hierarchy.

The specification of the protocol has been modeled in the SMV language using boolean variables to represent bus lines and control signals. SMV case statements have been written to determine the value of each signal, as defined by the futurebus protocol. In the SMV model each transition corresponds to one full transaction in the bus. Finally, the SMV model checker has been used to analyze the model compiled from the SMV code.

The analysis performed on this example concentrates on checking if the protocol maintains cache coherence, by verifying a number of properties. The first property states that if a cache has the only valid copy of some cache line, then no other caches should have valid copies of that line. The specification includes the formula

$$\mathbf{AG}(\texttt{p1.writable} \rightarrow \neg\texttt{p2.readable})$$

for each pair of caches `p1` and `p2`. The proposition `p1.writable` is true when `p1` is the only cache that has a valid copy of the cache line. Similarly, `p2.readable` is true when `p2` has one of possibly many valid copies.

Consistency is described by requiring that if two caches have copies of a cache line, then they agree on the data in that line:

$$\mathbf{AG}(\texttt{p1.readable} \wedge \texttt{p2.readable} \rightarrow \texttt{p1.data} = \texttt{p2.data})$$

Similarly, if memory has a copy of the line, then any cache that has a copy must agree with memory on the data.

$$\mathbf{AG}(\texttt{p.readable} \wedge \neg\texttt{m.memory\_line\_modified} \rightarrow \texttt{p.data} = \texttt{m.data})$$

The variable `m.memory_line_modified` is false when memory has an up-to-date copy of the cache line.

The last property expresses that it is always possible for a cache to get read or write access to the line.

$$\mathbf{AG}\ \mathbf{EF}\ \texttt{p.readable} \wedge \mathbf{AG}\ \mathbf{EF}\ \texttt{p.writable}$$

Several errors have been found in this analysis that were not previously known. For example, one counterexample showed an execution trace in which one processor had a cache line in the *shared unmodified* state, while a second one had the same cache line in the *exclusive modified* state. Another error showed a deadlock in the hierarchical configuration. Several different configurations have been verified, the largest one with three bus segments, eight processors, and over 10^{30} states.

4 Quantitative Algorithms

This section presents algorithms used for quantitative analysis and performance evaluation of models. First we describe algorithms that compute the minimum and maximum time delays between specified events. Then we show algorithms that determine the minimum and maximum number of times a given condition holds on any path from a set of starting states to a set of final states. Both algorithms have been used in the example presented subsequently.

4.1 Minimum and Maximum Delay Algorithms

In order to simplify the presentation, we must make some assumptions. All computations are performed on states reachable from a predefined set of initial states. We also assume that the transition relation is total. We consider the minimum delay algorithm first (figure 2). The algorithm takes two sets of states as input, *start* and *final*. It returns the length of (i.e. number of edges in) a shortest path from a state in *start* to a state in *final*. If no such path exists, the algorithm returns infinity. The function $T(S)$ gives the set of states that are successors of some state in S (it is computed using image computation [2], as discussed above). The function T, the state sets R and R', and the operations of intersection and union can all be easily implemented using BDDs.

```
proc minimum (start, final)
i = 0;
R = start;
R' = T(R) ∪ R;
while (R' ≠ R ∧ R ∩ final = ∅) do
        i = i + 1;
        R = R';
        R' = T(R') ∪ R';
if (R ∩ final ≠ ∅)
        then return i;
        else return ∞;
```

```
proc maximum (start, final)
i = 0;
R = TRUE;
R' = not_final;
while (R' ≠ R ∧ R' ∩ start ≠ ∅) do
        i = i + 1;
        R = R';
        R' = T^{-1}(R') ∩ not_final;
if (R = R')
        then return ∞;
        else return i;
```

Figure 2: Minimum and Maximum Delay Algorithms

The first algorithm is relatively straightforward. Intuitively, the loop in the algorithm computes the set of states that are reachable from *start*. If at any point, we encounter a state satisfying *final*, we return the number of steps taken to reach that state.

Next, we consider the maximum delay algorithm. This algorithm also takes *start* and *final* as input. It returns the length of a longest path from a state in *start* to a state in *final*. If there exists an infinite path beginning in a state in *start* that never reaches a state in *final*, the algorithm returns infinity. The function $T^{-1}(S')$ gives the set of states that are predecessors of some state in S' (i.e. $T^{-1}(S') = \{s \mid N(s, s') \text{ holds for some } s' \in S'\}$). We also denote by *not_final* the set of all states that are not in *final*. As before, the algorithm is implemented using BDDs, however, a backward search is required in this case.

4.2 Condition Counting Algorithms

In many situations we are interested not only in the length of a path from a set of starting states to a set of final states, but also in measures that depend on the number of states on the path that satisfy a given condition. For example, we may wish to determine the minimum (maximum) number of times a given condition holds on any path from starting to final states.

Both algorithms in this section take as input three sets of states: *start*, *cond* and *final*. The algorithms compute the minimum and the maximum number of states that belong to *cond*, over all finite paths that begin with a state in *start* and terminate upon reaching *final*.

To guarantee that the minimum (maximum) is well-defined, we assume that any path beginning in *start* must reach a state in *final* in a finite number of steps. This can be checked using the maximum delay algorithm described in the previous section. Finally, we ensure that all computations involve only reachable states, by intersecting *start* with the set of reachable states computed *a priori*.

To keep track at each step of the number of states in *cond* that have been traversed, we define a new state-transition system, in which the states are pairs consisting of a state in the original system and a positive integer. Thus, if the original state-transition graph has state set S, then the augmented state set will be $S_a = S \times \mathbb{N}$.

If $N \subseteq S \times S$ is the transition relation for the original state-transition graph, we define the augmented transition relation $N_a \subseteq S_a \times S_a$ as

$$N_a(\langle s, k\rangle, \langle s', k'\rangle) = N(s, s') \wedge (s' \in cond \wedge k' = k + 1 \vee s' \notin cond \wedge k' = k)$$

In other words, there will be a transition from $\langle s, k\rangle$ to $\langle s', k'\rangle$ in the augmented transition relation N_a iff there is a transition from s to s' in the original transition relation N and either $s' \in cond$ and $k' = k + 1$ or $s' \notin cond$ and $k' = k$. We also define T to be the function that for a given set $U \subseteq S_a$ returns the set of successors of all states in U. More formally, $T(U) = \{u' \mid N_a(u, u')$ holds for some $u \in U\}$. In the actual BDD-based implementation, an initial bound k_{max} can be selected to achieve a finite representation for k, and new BDD variables can be added dynamically if this bound is exceeded. The system is still finite-state because all paths we consider are finite and k is bounded by their maximum length.

proc *mincount* (*start, cond, final*)
current_min = ∞;
$R = \{\langle s, 1\rangle \mid s \in start \cap cond\} \cup \{\langle s, 0\rangle \mid s \in start \cap \overline{cond}\}$;
loop
    $Reached\_final = R \cap Final$;
    if $Reached\_final \neq \emptyset$ **then**
        $m = \min\{k \mid \langle s, k\rangle \in Reached\_final\}$;
        if $m < current\_min$ **then** $current\_min = m$;
    $R' = R \cap Not\_final$;
    if $R' = \emptyset$ **then return** *current_min*;
    $R = T(R')$;
endloop;

Figure 3: Minimum Condition Count Algorithm

The algorithm for computing the minimum count is given in figure 3. In the algorithm text, *Final* and *Not_final* denote the sets of states in *final* and $S - final$, paired with all possible values of k. More formally:

$$Final = \{\langle s, k\rangle \mid s \in final, k \in \mathbb{N}\} \quad \text{and} \quad Not\_final = \{\langle s, k\rangle \mid s \notin final, k \in \mathbb{N}\}$$

The algorithm uses R to represent the state set in S_a reached at the current iteration, while *Reached_final* and R' are its intersections with *Final* and *Not_final* respectively. Variable *current_min* denotes the minimum count for all previous iterations. The computation of the minimum value of k in a set of pairs $\langle s, k\rangle$ can be done by existentially quantifying the state variables (computing $K = \{k \mid \exists\langle s, k\rangle \in S\}$) and following the leftmost nonzero branch in the resulting BDD, provided an appropriate variable ordering is used.

At iteration i, the algorithm considers the endpoints of paths with i states. The reached states that belong to *final* are terminal states on paths that we need to consider. The minimum count for these paths is computed, using the counter component of the path endpoints, and the current value of the minimum is updated if necessary. For the reached states that do not belong to *final*, we continue the loop after computing their successors. If all reached states are in *final*, there are no further paths to consider and the algorithm returns the computed minimum.

Finally, we note that the algorithm for the maximum count has the same structure and can be obtained by replacing *min* with *max* and reversing the inequalities. Variants of both algorithms can be used to compute other measures that are a function of the number of states on a path that satisfy a given condition. For example, we can determine the minimum and the maximum number of states belonging to a given set $cond$ over all paths of a certain length l in the state space.

5 The PCI Local Bus

The PCI Local Bus [9, 10] is a high performance bus architecture that can have a data width of 32 or 64 bits. It has been designed by Intel to be used in its latest family of processors. Intel's goal is to offer a fast bus design at low cost that will accommodate current as well as future systems. PCI buses can be found in systems based on Alpha or Pentium processors. The majority of Pentium based systems manufactured today employ the PCI bus.

A typical PCI system can be seen in figure 4. The most important subsystems connected to the bus are the processor, a video controller, a SCSI controller, and an ISA bridge controller, which connects the PCI bus to a slower ISA bus. Modems, floppy disk controllers and other low speed components are connected to the ISA bus. Main memory and the secondary cache are connected directly to the processor using a PCI-memory-processor bridge. Other components can be added to the system. Usually, expansion slots are provided for this purpose.

Each of the subsystems shown above is allowed to request access to the bus and issue transactions. Slave subsystems are also supported; such subsystems respond to transactions, but do not issue them. A simplified PCI transaction can be seen in figure 5. A subsystem starts a transaction by asserting its request line `REQ`. It then waits until being granted the bus by the arbitration subsystem, which is indicated by the assertion of the `GNT` line. This phase is known as the *arbitration phase*. The next phase is the *bus acquisition phase*. The bus might not be idle when the new master is determined because the previous transaction may still be transferring data. Another transaction cannot be issued before all data has been transferred. The bus is idle whenever both signals `FRAME` and `IRDY` are deasserted in the same cycle, giving access of the bus to the new master. At this point the master asserts the `FRAME` signal, indicating the end of the bus acquisition phase and the beginning of a transaction. It also has to assert the signal `IRDY`, meaning that it is ready to send (or receive) data. The bus

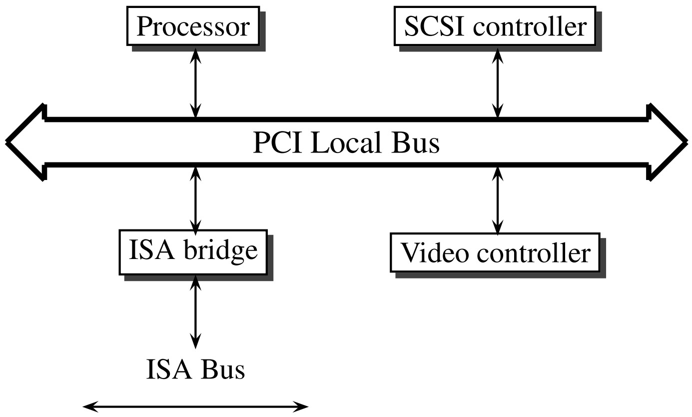

Figure 4: The PCI Local Bus

master has to wait for the target subsystem to respond by asserting its `TRDY` signal. This indicates that the target is ready to supply (or receive) data. The time interval between the start of a transaction and the assertion of the `TRDY` signal is called the *target response phase*. Data transfer starts when both `IRDY` and `TRDY` are asserted. One clock cycle before the end of the data transfer phase the `FRAME` signal is deasserted. At the next cycle both `IRDY` and `TRDY` are deasserted, and the bus becomes idle. In addition, transactions can be cancelled in various situations. This feature of the protocol is discussed in more detail later.

Arbitration in the PCI bus is implemented by a *two phase arbiter* as seen in figure 6. Each arbiter bank chooses among its incoming requests, and sends its decision to the following bank. The output of `Bank2` will be the new bus master. The decision is based on the `policy` signal, which can be set to *fixed priority* or *round-robin*. If all policies are set to the same value, the global arbitration policy will emulate either a fixed priority or a round-robin policy (Our analysis demonstrated that this is not always true. This result is discussed in the next section). However, mixed arbitration policies are possible by combining different policies in the banks.

Our model for the PCI bus follows the description above. Arbitration policies can be set to any possible combination, allowing mixed arbitration policies. However, in our model we must make some restrictions to the protocol described. For example, we must restrict the amount of data being transferred in one transaction. If this restriction is not implemented, no bounds on response time can be determined. In our model a single transaction can transfer between 1 and 16 cache lines of data. Our analysis will show how the information generated by this model can be used to determine the response time for models without this restriction. A similar approach has to be taken with the possibility of cancelling an ongoing transaction. Again, in order to prevent starvation, we must bound the number of times a transaction may be cancelled. Our final model for the PCI bus has 10^7

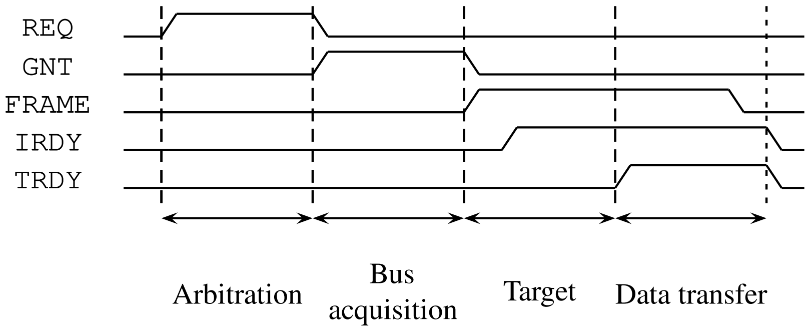

Figure 5: A transaction in the PCI Bus

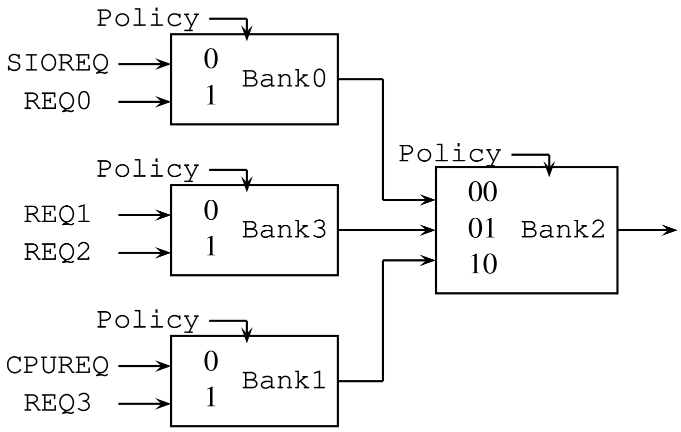

Figure 6: The PCI arbiter

reachable states out of a state space of 10^{18} states. The transition relation uses less than 10,000 BDD nodes, and the verification was completed in minutes.

6 Verification and Performance Analysis of the PCI Bus

Our analysis concentrates on the verification of issues such as transaction termination and arbitration fairness as well as on transaction performance. Being able to estimate the response time of a transaction is extremely important in any bus design, especially in one which has high performance as a primary goal. The bus data transfer rate and the overhead imposed by arbitration and communication protocols are examples of parameters involved in such an analysis. If those parameters cannot be determined, it will not be possible to design an optimized system that fully utilizes the available resources.

Moreover, the PCI bus is a good alternative for critical applications in which a bounded response time is vital. However, if the worst case response time of a transaction in the PCI bus hasn't been specified, such applications will most likely be implemented using other bus architectures. By bounding the worst time response of a transaction we hope to help application designers to evaluate the use of the PCI bus more accurately.

The correctness of the PCI bus protocol can be verified using the CTL model checker. For example, absence of starvation for bus access and transaction termination can be verified by the following formulas:

$$\mathbf{AG}(\mathtt{REQ} \rightarrow \mathbf{AF}\ \mathtt{GNT})$$

$$\mathbf{AG}(\mathtt{start\_transaction} \rightarrow \mathbf{AF}\ \mathtt{end\_transaction})$$

The properties above show that the response time of PCI transactions is bounded, but they give no indication of their performance. We will use the algorithms described in sections 4.1 and 4.2 to determine the response time for transactions. The results of our quantitative analysis also determine the correctness of the algorithm, for example, a transaction always finishes if its maximum response time is less than infinity.

In our performance analysis we will follow the structure of the protocol by computing the response time for each phase of the transaction separately. In this way we can have a better understanding of the behavior of the protocol. By computing the latency of each phase we are able to assert the efficiency of each step in the protocol and obtain the global behavior by adding individual figures. Results will be grouped into two categories, *total bus acquisition latency* and *total transaction latency*. The first category corresponds to the total time between a request being made on the bus and the subsystem actually being able to use the bus. The second category represents the total usage of the bus, that is, the time between asserting the `FRAME` signal until the end of data transfer. Table 7 shows the response times when the arbitration policy is set to round-robin in all banks and transaction cancelling is not allowed. Notice that in all cases discussed in this paper the latency for the data

Bus Master	Arbitration		Bus acquisition		Total bus acquisition		Target		Total transaction	
	min	max	min	max	min	max	min	max	min	max
ISA bridge	1	95	1	18	2	113	1	2	2	18
SCSI	1	95	1	18	2	113	1	2	2	18
Video	1	38	1	18	2	56	1	2	2	18
Processor	1	38	1	18	2	56	1	2	2	18

Figure 7: Response times for global round-robin policy

transfer phase varies between 1 and 16 clock cycles, there is no overhead associated with it. For that reason, this column will not be shown in the tables.

From the table above we can see two interesting properties of the system. The total transaction latency is at most 18 clock cycles, and in this case 16 clock cycles of data are transmitted. This means that once a master is able to use the bus, it can send data very efficiently. Another characteristic of the protocol is reflected on the bus acquisition times. The maximum of 18 cycles corresponds to one transaction. After being granted the bus the new master may have to wait for at most one more transaction to complete. This shows that once the bus is granted to a master, it will not be granted to another before the first one issues its transaction. Therefore no starvation can occur after a master is granted the bus. This property can be verified by:

$$\mathbf{AG}(\texttt{GNT} \rightarrow \mathbf{A}[\texttt{GNT}\ \mathbf{U}\ \texttt{FRAME}])$$

A more intriguing result can be seen in the arbitration latency results. The first two subsystems can take almost twice as long to access the bus as the others. In a round-robin environment, all subsystems should be granted equal usage of the resource, but this is not true in our example. By analyzing the execution traces produced by our tools we are able to determine the reason for the unfair access to the bus. The problem arises from the connection of the request lines to the arbiter as seen in figure 8. The ISA bridge and the SCSI controller are connected together to bank 0, while the video and the processor subsystems are alone in their banks. If bus traffic is high, the ISA bridge and the SCSI subsystems may have to wait for one another before their request reaches bank 2. Subsequently they may have to wait for subsystems connected to the other banks to execute before being granted the bus. In other words, they compete in both levels of arbitration, while the other subsystems only compete in the last level. This causes the worst time latency to be approximately twice as long for these subsystems. We can conclude from these results that two level arbitration *may* have a different behavior than an equivalent one level arbiter. In this case the problem is caused by an asymmetric connection of request lines.

We can also use these results to analyze the overhead imposed by the communication protocol on the transaction time. We have already seen that after asserting the `FRAME` signal there is an overhead of 2 clock cycles. This

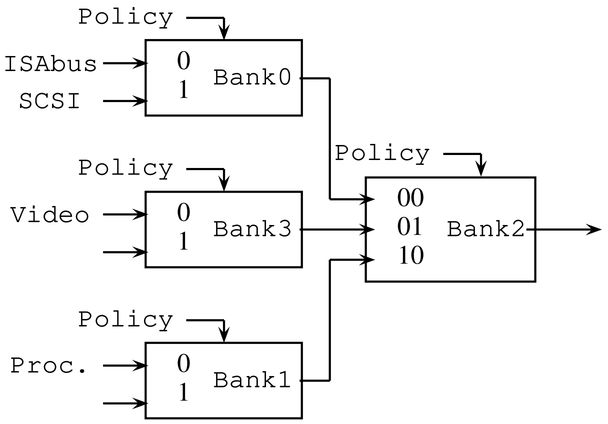

Figure 8: The PCI arbiter

overhead is independent of the transfer size. If a transaction is allowed to transfer more than 16 cache lines of data at once, the total utilization of the bus will increase. The designers of the bus can use this information to determine which is the best transfer size for a given system.

The following two formulas have been used to verify the above statements:

$$\mathbf{AG}(\texttt{FRAME} \rightarrow \mathbf{AF}_{\leq 2}(\texttt{state} = \texttt{DATA\_XFER}))$$

$$\mathbf{AG}((\texttt{state} = \texttt{DATA\_XFER}) \rightarrow \mathbf{A}[\texttt{state} = \texttt{DATA\_XFER}\ \mathbf{U}\ \texttt{end\_transaction}])$$

The first formula states that at most two cycles after the transaction starts, it will enter the data transfer phase. The second formula states that once a transaction is in the data transfer phase, it will continue in this phase until its end.

The overhead associated with arbitration can be computed in a similar way. It is more complex, however, because the arbitration latency depends not only on the transaction time, but also on the number of active request lines. We use the condition counting algorithms to uncover more details about this problem. We compute the number of transactions issued on the bus between the time a master requests access and the time it is granted the bus. Up to 5 transactions can be issued during this period for the ISA bridge and the SCSI subsystems, and up to 2 transactions can be issued for the video and processor subsystems. Total transaction time for each of these intermediate transactions is 18 clock cycles. By comparing the total effective data transfer time with the maximum arbitration time, we can see that each intermediate transaction has an arbitration time of one clock cycle. These results are also valid for the video and processor subsystems. We can conclude that the arbitration latency can be computed by the formula: Arbitration_Latency = n * (Transaction_Latency + 1), where n is the maximum number of intermediate transactions that can be issued between a request and the corresponding grant (computed with the condition counting algorithms). This formula does not depend on maximum data transfer size.

Bus Master	Arbitration		Bus acquisition		Total bus acquisition		Target		Total transaction	
	min	max	min	max	min	max	min	max	min	max
ISA bridge	1	19	1	18	2	37	1	2	2	18
SCSI	1	∞	1	18	2	∞	1	2	2	18
Video	1	∞	1	18	2	∞	1	2	2	18
Processor	1	∞	1	18	2	∞	1	2	2	18

Figure 9: Response times for global fixed priority policy

The above results assume a global round-robin policy. The behavior of the system under a fixed priority arbitration policy has also been studied and the results can be seen in table 9. The ISA bridge is the highest priority subsystem on the bus. Its response time is much lower in the fixed priority configuration than in the round-robin one. However, all other subsystems may starve, since the ISA bridge can continuously issue transactions. Notice that the arbitration time, but not the transaction time, is affected by the arbitration policy. These response times can be used by the designer to check if the performance of the PCI bus is adequate for a critical application. Other combinations of arbitration policies are possible, but are not presented here for the sake of brevity.

The model described above allows a detailed analysis of the behavior of the PCI bus protocol. Some features of the actual bus, such as parity or data width, have been abstracted from our model, since they do not affect the timing of transactions. However, there are other features that do affect timing such as the possibility of a transaction being cancelled. Errors on the bus may occur, the target may be slow, or unable to produce the data. For example, a transaction requesting data from the ISA bus will most likely experience a long delay, simply because of the relative speeds of the ISA and PCI buses. In the model described above this feature has been abstracted out by the assumption that the target of a transaction responds immediately. A more realistic model that allows transactions to be cancelled has also been implemented.

In order to account for long delay responses and aborted transactions we introduce the concept of transaction cancellation in our model. Transactions may be cancelled any time they are in progress. Transaction cancellations model the fact that in the actual PCI bus whenever a target is unable to answer for a long time, it aborts the transaction, which is reissued later. We model this situation by cancelling the transaction and restarting it immediately by issuing another request. However, reissuing the transaction immediately would not correctly model the response time of a very slow target. To accommodate this situation, in our model a cancelled transaction is restarted as many times as necessary to accommodate the target response time. Using the algorithms described we compute the overhead caused by cancelling and restarting a transaction, and use this result to determine the number of retries for the response delay of a given target.

Bus Master	Arbitration		Bus acquisition		Total bus acquisition		Target		Total transaction	
	min	max	min	max	min	max	min	max	min	max
ISA bridge	1	95	1	18	2	113	1	6	2	132
SCSI	1	95	1	18	2	113	1	6	2	132
Video	1	38	1	18	2	56	1	6	2	75
Processor	1	38	1	18	2	56	1	6	2	75

Figure 10: Response times for global round-robin policy, maximum one cancel

Moreover, unlimited cancellations may cause starvation. Therefore, in order to compute the worst time response, we must limit the number of cancellations allowed. A cancellation brings the bus to the idle state, as can be verified by the following CTL formula:

$$\mathbf{AG}(\texttt{ABORT} \rightarrow \mathbf{AX}\ \texttt{BUS\_IDLE})$$

As a consequence, consecutive cancellations have the same behavior, because a cancellation brings the system into the same state as before the transaction. Therefore, the total overhead caused by n cancellations is n times the overhead of a single cancellation. Therefore, it suffices to consider the situation in which at most one cancellation occurs. The results for a global round-robin arbitration policy in the presence of at most one transaction cancellation are presented in table 10.

In this table we can see that arbitration latency is not affected by transaction cancellations. The reason is that whenever a transaction is cancelled the current bus master releases the bus and becomes last in the round-robin queue. On the other hand, total transaction latency increases significantly. The execution trace of the transaction with the worst latency shows the following sequence of events (for the ISA bridge subsystem):

1. A transaction starts but is cancelled just before completion, after 17 clock cycles.

2. Another request is made to complete it in the next cycle (one extra clock cycle).

3. An arbitration sequence of 79 cycles follows.

4. A bus acquisition phase starts, taking 17 clock cycles.

5. The transaction starts again, completing in 18 cycles.

The arbitration sequence appearing in item 3 is the same as in the worst case, except that the request is made when *the bus is already idle* because of the cancellation. The difference of 16 clock cycles corresponds to one maximum data transfer phase done by another bus master, as shown by the counterexample for the worst case arbitration latency (not presented for brevity). The total delay caused by the first three items is the equivalent of a worst case arbitration latency plus two clock cycles, caused by the cancellation. A bus acquisition phase and a transaction latency phase, in which no cancellation occurs, account for the last 35 cycles. We can see then that the overhead imposed by a transaction cancellation consists of a worst case arbitration latency, a maximum bus acquisition phase, a maximum transaction latency (without cancellations) and one extra clock cycle. Again, this formula applies for the video and processor subsystems. These results may be used to estimate the performance of an implementation of the PCI in the presence of transaction aborts. The formula derived gives the overhead for one transaction cancellation, and can be extended to many cancellations as well. In this manner, the worst response time in various configurations of the system can be computed.

To summarize our results, we have been able to:

- Model the PCI Local bus protocol and verify its correctness. In the round-robin case no starvation of subsystems occur, and transactions always finish, even in the presence of limited cancellations.
- Determine the minimum and maximum latencies for each phase of the protocol, and show which phases are affected by changes in the parameters (such as arbitration policy and presence of cancellations).
- Compute response times independent of specific values for the data transfer phase.
- Determine response time in the presence of limited transaction aborts using the condition counting algorithms described.

These results allow the designers of the protocol to understand its actual behavior and how this behavior changes when parameters of the system are modified. We believe that this is valuable information when verifying and optimizing a new hardware system. This example shows that our method can be used to analyze the performance of modern hardware designs that have very complex behavior. It can help improve the reliability of new products and increase the efficiency of the design process.

7 Conclusion

Model checking is a well established technology for formal verification. Using this method we have been able to verify systems of industrial complexity, such as the Futurebus+ cache coherence protocol. This analysis discovered errors on the protocol that were not known before.

We have also extended model checking techniques to allow a quantitative analysis of models as well as performance evaluation. This paper presents algorithms to compute minimum and maximum path lengths as well as the minimum and maximum number of times an event occurs on all paths from a set of start states to a set of final states. The analysis of the PCI Local bus demonstrates the power of this technique. The PCI is a high-performance bus design used in most Pentium processor based systems. By analyzing its performance we have shown that our techniques can be used in complex industrial designs.

The measurements produced by these algorithms can be used to analyze design decisions before the system is actually implemented. In the PCI bus example, the description of the hardware can easily be modified to model different arbitration policies and different data transfer sizes. This flexibility allows designers to fine-tune system parameters in order to maximize efficiency.

The method presented can help determining the correctness of computer systems, as well as evaluate their performance. It is versatile enough to enable several types of analysis to be performed, and efficient enough to be used in complex modern industrial designs. We believe they can be of significant help in designing correct applications, as well as in reducing costs of the development process.

References

[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–91, 1986.

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 46–51, 1990.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 403–7, 1990.

[4] S. V. Campos. The priority inversion problem and real-time symbolic model checking. Technical Report CMU-CS-93-125, Carnegie Mellon University, 1993.

[5] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[6] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In L. Claesen, editor, *International Symposium on Computer Hardware Description Languages and their Applications*. North-Holland, April 1993.

[7] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In *Lecture Notes in Computer Science*, volume 531, pages 136–45. Springer–Verlag, 1990.

[8] IEEE Computer Society. *IEEE Standard for Futurebus+ – Logical Protocol Specification*, March 1992. IEEE Standard 896.1–1991.

[9] Intel Corporation. *82378 System I/O (SIO) - PCI Local Bus*, 1993.

[10] Intel Corporation. *PCI Local Bus Specification*, 1993.

[11] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.